More RAII

EECS 211

Winter 2019

Road map

- A reference-counted string class
- Introduction to smart pointers

The `Shared_string` class

Reference counting idea

- Keep a count of how many objects point to the same data
- Increment count in copy constructors
- Decrement count in destructors, and free data if count reaches 0

Essence of `class Shared_string` (1/5)

```
class Shared_string
{
public:
    Shared_string();
    Shared_string(string_view);
    Shared_string(Owned_string&&);
    Shared_string(Shared_string const&);
    Shared_string(Shared_string&&);

    Shared_string& operator=(string_view);
    Shared_string& operator=(Owned_string&&);
    Shared_string& operator=(Shared_string const&);
    Shared_string& operator=(Shared_string&&);

    ~Shared_string();
    ...
};
```

Essence of `class Shared_string` (2/5)

```
class Shared_string
{
    ...
private:
    void increment_();
    void decrement_();
    void pod_assign_(Shared_string const&);

    size_t*     ref_count_; // owned, >= 1
    char*       data_;      // owned
    string_view view_;      // points into data_[]
};
```

Essence of `class Shared_string` (3/5)

```
void Shared_string::increment_()
    { if (ref_count_) ++*ref_count_; }

void Shared_string::decrement_()
{
    if (!ref_count_ || --*ref_count_) return;
    delete ref_count_;
    delete [] data_;
}

void Shared_string::pod_assign_(
    Shared_string const& o)
{
    ref_count_ = o.ref_count_;
    data_      = o.data_;
    view_      = o.view_;
}
```

Essence of `class Shared_string` (4/5)

```
Shared_string::Shared_string(
    Shared_string const& o)
        : ref_count_(o.ref_count_)
        , data_(o.data_)
        , view_(o.view_)
{
    increment_();
}

Shared_string::~Shared_string()
{
    decrement_();
}
```

Essence of `class Shared_string` (5/5)

```
Shared_string&
Shared_string::operator=(Shared_string const& o)
{
    o.increment_();
    decrement_();
    pod_assign_(o);
    return *this;
}

Shared_string&
Shared_string::operator=(Shared_string&& o)
{
    decrement_();
    pod_assign_(o);
    o.pod_assign_({});
    return *this;
}
```

Smart pointers

Smart pointers have destructors

It's a class that you use like a pointer, but has a destructor:

```
#include <memory>

TEST_CASE("unique_ptr<int>")
{
    std::unique_ptr<int> p;
    CHECK( p == nullptr );

    p = std::make_unique<int>(3);
    CHECK( p != nullptr );
    CHECK( *p == 3 );

    ++*p;
    CHECK( *p == 4 );
} // deleted here
```

`std::unique_ptr` is *move-only*

The copy constructor and copy assignment operator are *deleted*

```
TEST_CASE("move, don't copy")
{
    std::unique_ptr<int> p, q;

    p = std::make_unique<int>(3);

    q = p;  // type error



}
```

`std::unique_ptr` is *move-only*

The copy constructor and copy assignment operator are *deleted*

```
TEST_CASE("move, don't copy")
{
    std::unique_ptr<int> p, q;

    p = std::make_unique<int>(3);

    q = std::move(p);

    CHECK( *q == 3 );
    CHECK( p == nullptr );
}
```

A class for tracking destruction (1/2)

```
class tracker
{
public:
    class guard;
    using log_t = std::vector<int>;
    using ptr_t = std::unique_ptr<guard>;

    log_t const& log() const { return log_; }

    ptr_t operator()(int tag)
    {
        return ptr_t(new guard(log_, tag));
    }

private:
    log_t log_;
};
```

A class for tracking destruction (2/2)

```
class tracker::guard
{
public:
    ~guard() { log_.push_back(tag_); }

    guard(guard const&) = delete;
    guard& operator=(guard const&) = delete;

private:
    friend tracker;
    guard(log_t& log, int tag)
            : log_(log), tag_(tag) { }

    log_t& log_;
    int    tag_;
};
```

Using the tracker class

```
using log_t = tracker::log_t;

TEST_CASE("tracking␣unique_ptr")
{
    tracker track;

    track(3);
    CHECK( track.log() == log_t{3} );

    {
        tracker::ptr_t p1 = track(5);
        tracker::ptr_t p2 = track(7);
        CHECK( track.log() == log_t{3} );
    }

    CHECK( track.log() == log_t{3, 7, 5} );
}
```

Aside: `std::shared_ptr` is reference counted

Like `Shared_string`.

```
TEST_CASE("std::shared_ptr<int>")
{
    std::shared_ptr<int> p =
        std::make_shared<int>(8);
    CHECK( p.use_count() == 1 );

    std::shared_ptr<int> q = p;
    CHECK( p.use_count() == 2 );

    *q *= 2;
    CHECK( *p == 16 );

    q = nullptr;
    CHECK( p.use_count() == 1 );
}
```

The essence of `std::unique_ptr`

```
class unique_ptr
{
    T* raw_;

public:
    explicit unique_ptr(T* raw = nullptr)
            : raw_(raw) { }

    unique_ptr(unique_ptr const&) = delete;

    unique_ptr(unique_ptr&& o) : raw_(o.raw_)
    { o.raw_ = nullptr; }

    T& operator*() const
    { return *raw_; }
};
```

The essence of `std::unique_ptr`

```
template <class T>
class unique_ptr
{
    T* raw_;

public:
    explicit unique_ptr(T* raw = nullptr)
            : raw_(raw) { }

    unique_ptr(unique_ptr const&) = delete;

    unique_ptr(unique_ptr&& o) : raw_(o.raw_)
    { o.raw_ = nullptr; }

    T& operator*() const
    { return *raw_; }
};
```

– Next: Figuring Stuff Out —